featuremonkey Documentation

Release 0.3.1

Hendrik Speidel

September 18, 2016

Contents

> `featuremonkey` is a tool to support *feature oriented programming (FOP)* in python.

`featuremonkey` is a tiny library to enable feature oriented programming (FOP) in Python. Feature oriented software development(FOSD) is a methodology to build and maintain software product lines. Products are composed automatically from a set of feature modules and may share a set of features and differ in others.

There are multiple definitions of what a feature really is. Here, we use the definition of Apel et al.:

> A feature is a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option *[ALMK]* .

When trying to modularize software-systems to acheive reusability, components come to mind. However, large components are often very specific which limits their reuse; many small components often make it necessary to write a large amount of glue code to integrate them.

So components are nice — but it feels like there is something missing.

Features provide an additional dimension of modularity by allowing the developer to encapsulate code related to a specific concern that is scattered across multiple locations of the codebase into so-called *feature modules*. Products can then be composed automatically by selecting a set of these feature modules.

Common approaches to FOSD are the use of generative techniques i.e. statically composing a product's code and other artefacts as part of the build process e.g. FeatureHouse, the use of specialized programming languages with feature support e.g. FeatureC++, or by making features explicit using IDE support e.g. CIDE.

`featuremonkey` implements feature composition by using monkeypatching i.e. structures are dynamically modified at runtime.

CHAPTER 1

Fun facts on featuremonkey for FOSD people

- dynamic feature binding at startup time or later
- no unbind
- in Python everything is an object — `featuremonkey` composes objects
- function/method refinements are implemented as delegation layers(wrappers wrapping wrappers wrapping ...)
- uses monkeypatching to bind features — dynamic program modification

The central operation exposed by `featuremonkey` is `compose`. It applies transformations.

CHAPTER 2

Feature Oriented Software Development

CHAPTER 3

Getting started

3.1 Installation

3.1.1 Installing the latest stable version

Make sure you have `pip` installed. `featuremonkey` can then be installed using the following command:

```
pip install featuremonkey
```

3.1.2 Installing the development version

To get the development version of `featuremonkey` directly from github, use:

```
pip install git+https://github.com/henzk/featuremonkey.git#egg=featuremonkey
```

You can check by importing `featuremonkey` from a python prompt. If you don't see an error, everything should be ok.

CHAPTER 4

featuremonkey Reference

4.1 featuremonkey Reference

4.1.1 Feature Composition

A feature bundles *introductions* and *refinements* to the codebase. Introductions are additions to the codebase, refinements allow certain modifications of existing code.

We will use *structure transformation* or transformation as the more generic term, when referring to introductions and refinements (Often, refinement is used instead, because technically an introduction is a form of refinement — transformation is used here just to avoid confusion).

The process of actually applying the introductions and modifications of a given feature is called *feature binding*.

Feature Composition is the stepwise binding of a *feature selection* — a specified set of features in a specified order.

4.1.2 Feature Layout

Features are represented as Python packages. The feature name is defined as the fully qualified name of the package.

The package needs to define a module called `feature`, which must define a function `select(composer)`.

This function is called by the composer to bind the feature. Its purpose is to apply the structure transformations defined by the feature using the composer, which gets passed in as sole argument.

File Structure

```
myfeature/
    __init__.py
    feature.py
```

Above, mandatory files to specify a feature called `myfeature` are listed. Additionally, features may contain other modules, subpackages and also non-Python files.

Note: `__init__.py` is needed to mark the directory as a python package.

```
#myfeature/feature.py

def select(composer):
    #bind myfeature by applying necessary transformations
```

```
    #apply transformation myfeature.mymodule to basefeature.mymodule
    from . import mymodule
    import basefeature.mymodule
    composer.compose(mymodule, basefeature.mymodule)
```

Here, the example of a `feature` module inside a feature package is given.

`select` is mandatory, but may be empty if there are no transformations to apply, e.g. in case of the base feature.

In the following, the different types of structure transformations offered by the composer are described.

4.1.3 Feature Structure Trees

featuremonkey recognizes python packages, modules, classes, functions and methods as being part of the FST.

4.1.4 FST Declaration

FSTs are declared as modules or classes depending on the preference of the user. modules and classes can be mixed arbitrarily.

Note: when using classes, please make sure to use new style classes. Old style classes are completely unsupported by featuremonkey - because they are old and are removed from the python language with 3.0. To create a new style class, simply inherit from `object` or another new style class explicitely.

FSTs specify introductions and refinements of structures contained in the global interpreter state. This is done by defining specially crafted names inside the FST module/class.

FST Introduction

Introductions are useful to add new attributes to existing packages/modules/classes/instances.

An introduction is specified by creating a name starting with `introduce_` followed by the name to introduce directly inside the FST module/class. The attribute value will be used like so to derive the value to introduce:

- If the FST attribute value is not callable, it is used as the value to introduce without further processing.
- If it is a callable, it is called to obtain the value to introduce. The callable will be called without arguments and must return this value.

Example:

```
class TestFST1(object):
    #introduce name ``a`` with value ``7``
    introduce_a = 7

    #introduce name ``b`` with value ``6``
    def introduce_b(self):
        return 6

    #introduce method ``foo`` that returns ``42`` when called
    def introduce_foo(self):
        def foo(self):
            return 42

        return foo
```

Warning: Names can only be introduced if they do not already exist in the current interpreter state. Otherwise `compose` will raise a `CompositionError`. If that happens, the product may be in an inconsistent state. Consider restarting the whole product!

FST Refinement

Refinements are used to refine existing attributes of packages/modules/classes/instances.

An introduction is specified much like an introduction. It is done by creating a name starting with `refine_` followed by the name to refine directly inside the FST module/class. The attribute value will be used like so to derive the value to introduce:

- If the FST attribute value is not callable, it is used as the refined value without further processing. **This is a replacement**
- If it is a callable e.g. a method, it is called to obtain the refined value. The callable will be called with the single argument `original` and must return this value. `original` is a reference to the current implementation of the name that is to be refined. It is analogous to `super` in OOP.

Example:

```
class TestFST1(object):
    #refine name ``a`` with value ``7``
    refine_a = 7

    #refine name ``b`` with value ``6``
    def introduce_b(self, original):
        return 6

    #refine method ``foo`` to make it return double the value of before.
    def refine_foo(self, original):
        def foo(self):
            return orginal(self) * 2

        return foo
```

Note: when calling `original` in a method refinement(for both classes and instances), you need to explicitely pass `self` as first parameter to `original`.

Warning: Names can only be refined if they exist in the current interpreter state. Otherwise `compose` will raise a `CompositionError`. If that happens, the product may be in an inconsistent state. Consider restarting the whole product!

4.1.5 FST nesting

FSTs can be nested to refine nested structures of the interpreter state. To create a child FST node, create a name starting with `child_` followed by the nested name. The value must be either a FST class or instance or a FST module. As an example, consider a refinement to the `os` module. We want to introduce `os.foo` and also refine `os.path.join`. We could do this by composing a FST on `os` to introduce `foo` and then composing another FST on `os.path` that refines `join`. Alternatively, we can use FST nesting and specify it as follows:

```
class os(object):
    introduce_foo = 123
    class child_path(object):
        def refine_join(self, original):
            def join(*elems):
                return original(elems)
            return join
```

Got it?

4.1.6 FST Composition

featuremonkey.**compose**(*self*, **things*)
compose applies multiple fsts onto a base implementation. Pass the base implementation as last parameter. fsts are merged from RIGHT TO LEFT (like function application) e.g.:

class MyFST(object): #place introductions and refinements here introduce_foo = 'bar'

compose(MyFST(), MyClass)

featuremonkey.**compose_later**(*self*, **things*)
register list of things for composition using compose()

compose_later takes a list of fsts. The last element specifies the base module as string things are composed directly after the base module is imported by application code

4.1.7 Product Selection

featuremonkey.**select**(*self*, **features*)
selects the features given as string e.g passing 'hello' and 'world' will result in imports of 'hello' and 'world'. Then, if possible 'hello.feature' and 'world.feature' are imported and select is called in each feature module.

featuremonkey.**select_equation**(*self*, *filename*)
select features from equation file

format: one feature per line; comments start with #

Example:

```
    #this is a comment
    basefeature
    #empty lines are ignored

    myfeature
    anotherfeature
```

4.1.8 Import Guards

class featuremonkey.importhooks.**ImportGuardHook**
Import Hook to implement import guards.

In Python imports can have side-effects and import order may be relevant. When using featuremonkey, it is important to compose the product before making references to it. Otherwise, you could end up with a reference to a module/class/object that has only been composed partially. This may introduce subtle bugs that are hard to debug.

Using an import guard, you can enforce that a module cannot be imported until the import guard on that module is dropped again. Importing a guarded module results in an ImportGuard exception being thrown. Usually, you don't want to catch these: better fail during the composition phase than continuing to run a miscomposed program.

The existance of the import hook is considered an implementation detail. The public API to import guards are `featuremonkey.add_import_guard` and `featuremonkey.remove_import_guard`.

`featuremonkey.`**`add_import_guard`**(*module_name*, *msg=''*)
Until the guard is dropped again, disallow imports of the module given by `module_name`.

If the module is imported while the guard is in place an `ImportGuard` is raised. An additional message on why the module cannot be imported can optionally be specified using the parameter `msg`.

If multiple guards are placed on the same module, all these guards have to be dropped before the module can be imported again.

`featuremonkey.`**`remove_import_guard`**(*module_name*)
drop a previously created guard on `module_name` if the module is not guarded, then this is a no-op.

Example:

```
>>> import featuremonkey
>>> featuremonkey.add_import_guard('django')
>>> import django
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "featuremonkey/importhooks.py", line 160, in load_module
    + (self._guards[module_name][-1] or module_name)
featuremonkey.importhooks.ImportGuard: Import while import guard in place: django
>>> featuremonkey.remove_import_guard('django')
>>> import django
>>>
```

First an import guard is created for the package `django`. Then, we try to import it and an `ImportGuard` is raised. After we remove the guard again, we can import the package without an error.

4.1.9 Utilities

`featuremonkey.`**`get_features_from_equation_file`**(*filename*)
returns list of feature names read from equation file given by `filename`.

format: one feature per line; comments start with #

Example:

```
    #this is a comment
    basefeature
    #empty lines are ignored

    myfeature
    anotherfeature
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

CHAPTER 6

Changelog

6.1 Changelog

0.3.1

- support refinement of methods decorated with `staticmethod` and `classmethod`
- improved docs
- fixed broken examples

0.3.0

- better error messages
- feature.py is now mandatory for features
- compose_later also accepts transformations specified as strings (these are assumed to be module names and will be imported at composition time)
- function refinements that don't carry docstrings now use the docstring of their original

0.2.2

- added `get_features_from_equation_file` to public API
- added import guards
- split into multiple files
- **backwards incompatible change:** signatures of `feature.select` functions need to be changed from `feature.select` to `feature.select(composer)`.

0.2.1

- more docs
- raises `CompositionError` consistently

0.2

- first release on PYPI
- composer is now class based
- fixes compose_later composition order
- initial docs

0.1

- initial version

[ALMK] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebra for Features and Feature Composition. In Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST), volume 5140 of Lecture Notes in Computer Science, pages 36–50. Springer-Verlag, 2008.

A

C

G

I

R

S